Design Rationale Sprint 2 - Alex Ung
Explain two key classes (not interfaces) that you have included in your design and provide your reasons why you decided to create these classes. Why was it not appropriate for them to be methods?
Classes:

MovementManager
The MovementManager class is very important in my design for the fiery dragons game. Anything that requires movement will be passed through this class. This class has 3 methods in total excluding the getInstance method. So in the fiery Dragon board game, movement can be made by Dragon Tokens. The first thing that must be considered before a DragonToken can move, is can the DragonToken legally move to a new position according to the game rules.
The game rules specify that a Dragon Token can move to a new position as long as:

- There is not another Dragon Token on the new position
- If the movement is backwards, is the DragonToken inside of their cave already?

The second method is named updatePosition, so this handles all of the nitty gritty things when it comes to updating the player's position. It does not update the UI, but it handles the internal game logic. It determines how many positions to move depending on if the DragonToken is still in their cave or not and it takes into account backwards movements.

For when DragonCards are implemented, the movementManager will leverage a third method within itself which is the isMatch() method. This will be where the logic for handling matching dragon cards and square creatures which will determine whether the movement can be made.

The reason why I did not have this class as a set of methods inside the player or the board for example, is because I think that there are too many responsibilities related to the movement logic for the game. If I had this class merged with another class like the DragonToken class, the DragonToken class would have too many responsibilities and hence breaking the single responsibility principle. By having this movement manager class I have been able to encapsulate everything related to movement in this class. This will also make it easier if I want to add different kinds of movements in the future because there is a standalone class that manages it which would respect the open/closed principle.

BoardArray class
The board array class is the glue for the creation of the board. The BoardArray holds everything related to the creation of the board. I wanted to have the logic for the board all in one place which makes it easier to access different parts of the board. So for the BoardArray class, it will add VolcanoCards and shuffle them to be placed on the board, it will also add the dragon cards to the board. The reason why this class cannot be a method is because it would add another responsibility to another class and I wouldn't want to have access to information that is not relevant to the actual board itself when I am accessing the board from another

class. This class aims to be in line with the single responsibility and is a part of the chain of responsibilities design pattern that will be discussed later on in the rationale.

Explain two key relationships in your class diagram, for example, why is something an aggregation not a composition?

volcanoCard -> Square Composition relationship

So in the case of my volcanoCard class, my interpretation of the game makes me think that because volcano cards are pieces that are placed by the BoardArray, why should the BoardArray have to take care of the squares that are on each volcanoCard. So I've made the VolcanoCard class have a composition relationship with the square class. In the context of the game itself, if there are no volcano cards, then there wouldn't be any squares. So it wouldn't be aggregation because a square cannot exist independently of a volcano card.

DragonToken -><- DragonTokenPanel

So this relationship between the two classes represents a circular association. In this scenario I do not think that having circular dependencies is always bad. In this case I don't think it is a bad design purely because in order to move dragon tokens i'll need to have access to the specific panel that the dragonToken has been allocated. The same goes for the dragonTokenPanel, in the case of the UI, in order to move a dragonTokenPanel I'll need to know the position of the DragonToken that the panel is assigned. So there are some things that I have considered in regard to this relationship, one of the solutions is to combine the two classes together, but I think that's bad because there would be no separation between the UI and the backend which is not encouraged. I didn't want to use an interface for this scenario because of the Dependency Inversion principle of wanting to have classes only depend on abstractions instead of interfaces. The other workaround would be to pass instances of each other through method parameters but that allows for different dragon tokens to be passed through to a different panel.

Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?

I used it, but only in applications where I saw extensions to be applicable. For example, I used inheritance on the DragonToken class because a DragonToken is an actor and there could definitely be extensions that could involve the introduction of actors that could do different things. To give an example, it would be possible to have multiple different types of dragonTokens that have different abilities like a Mischievous wizard token that can activate obstacles or give riddles to a player to set them back a few squares.

The other case where I used inheritance was with the action class. For now there is really only one action that a player can take which is flipping a dragon card but what if there was an introduction of landing on a square which allows the player to choose to pick up a card (from a deck of cards) that acts like a mystery card. This reduces a lot of the repeated code that is seen with classes that have similar attributes and methods.

I chose to not use inheritance in regards to the creatures because they don't hold any information and are only used as an identifier to compare with the creatures of a dragon card. In my opinion, adding another layer of abstraction for the creatures would increase the complexity of the application and would therefore not be worth it in this case.

Explain how you arrived at two sets of cardinalities, for example, why 0..1 and why not 1…2?
In a more general sense, the way that I produce cardinalities is mainly through the logic of how the game is played. The game has a specific set of rules that should be followed for most of the cardinalities. So for example in the case of the BoardArrray and the Game class, there can only ever be one board in the game and you can't have 1 without the other to play the game so their cardinality relationship is 1-1. In the context of another relationship like volcano cards having a relationship with squares 1 to 3 respectively it's because for each volcano card that exists there can only be 3 squares according to the game rules. In the case these rules can change the cardinality can be changed accordingly. There are cases where cardinality is determined through code only, but in my case most of those are always 1 to 1 because that makes the most sense.

If you have used any Design Patterns in your design, explain why and where you have applied them. If you have not used any Design Patterns, list at least 3 known Design Patterns and justify why they were not appropriate to be used in your design.

So in my design I have multiple instances where I have applied the singleton pattern. The classes that I have used singleton include: MovementManager, PlayerManager,BoardArray. The reason I have used these patterns for these classes is because each of these classes have a set of resources that need to be accessed across multiple classes and to make sure that everything remains consistent, having one instance makes it a lot easier. Since pretty much all of the attributes and methods don't change, having one instance also saves memory through unnecessary instantiations across the system.

In the case of MovementManager, I wanted there to only be one class that controls movement across all actors in the game. This way each actor would only have to access a single instance instead of each having their own instance which would be redundant.

The PlayerManager class is a global access point to know how many dragon tokens there are in the game and keeps a global track for knowing who's turn it is to make a move.

For the BoardArray, Singleton was used because again, in order to have access to the squares on the board, the creatures that inhabit them as well as the dragon cards, a lot of different classes will require access to this information. For example, the movement class, the player manager class, the game class and the window panel class.

In terms of not using design patterns in my design, I did consider a few, one of which was the use of the abstract factory design pattern.

The abstract factory class involves the creation of families of classes without having to specify their concrete classes. The scenario in which I thought it could be implemented would be for the creature logic. My thought was that yes I probably could have an abstract factory to create the creatures for me and while yes I do think that it could be implemented in this scenario as there are many creatures of similar family that would need to be created for the board to be functional but I believe that it adds another layer of complexity to the code which is unnecessary for the current design. I also think that because I have had no experience using this design pattern it would be difficult for me in my situation to implement such a design pattern correctly on the first go. After some evaluation I stuck with the implementation of the enumeration class because there would be significantly fewer classes and instances having to be created and for the game itself, the creatures do not serve a purpose other than being identifiers to determine how a DragonToken moves. So I have taken the trade of having a few more if statements over a layer of complexity.

Another design pattern that I thought about was the builder design principle. The aim of the builder design pattern is to lessen the complexity of creating objects that need to have a lot of different information to be passed through. For example if a restaurant wants to serve multiple dishes of food but then also there are different styles of creating these foods (different origins) which would make the code very messy if you continue to have to create a bunch of different but similar classes to do this. In the context of the Fiery Dragons game I believe that this scenario couldn't be applied. There is no instance that I can think of that would require a solution to the problem that the design pattern aims to solve. This is probably due to the overall simplicity of how the FieryDragons game works.